

JP 0097451

日 本 国 特 許 庁

PATENT OFFICE
JAPANESE GOVERNMENT

JP714 U.S. PRO
09/708159
11/08/00

別紙添付の書類に記載されている事項は下記の出願書類に記載されている事項と同一であることを証明する。

This is to certify that the annexed is a true copy of the following application as filed with this Office.

出 願 年 月 日

Date of Application:

1999年11月17日

出 願 番 号

Application Number:

平成11年特許願第326990号

出 願 人

Applicant (s):

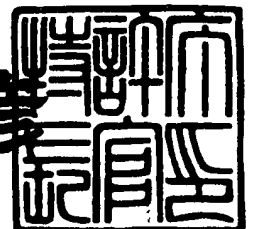
インターナショナル・ビジネス・マシーンズ・コーポレイション

CERTIFIED COPY OF
PRIORITY DOCUMENT

2000年 3月24日

特許庁長官
Commissioner,
Patent Office

近 藤 隆 彦



出証番号 出証特2000-3021151

【書類名】 特許願

【整理番号】 JA999097

【あて先】 特許庁長官 殿

【国際特許分類】 G06F 9/44

【発明者】

【住所又は居所】 神奈川県大和市下鶴間 1 6 2 3 番地 1 4 日本アイ・ピー・エム株式会社 東京基礎研究所内

【氏名】 安江 俊明

【発明者】

【住所又は居所】 神奈川県大和市下鶴間 1 6 2 3 番地 1 4 日本アイ・ピー・エム株式会社 東京基礎研究所内

【氏名】 緒方 一則

【発明者】

【住所又は居所】 神奈川県大和市下鶴間 1 6 2 3 番地 1 4 日本アイ・ピー・エム株式会社 東京基礎研究所内

【氏名】 石崎 一明

【発明者】

【住所又は居所】 神奈川県大和市下鶴間 1 6 2 3 番地 1 4 日本アイ・ピー・エム株式会社 東京基礎研究所内

【氏名】 小松 秀昭

【特許出願人】

【識別番号】 390009531

【住所又は居所】 アメリカ合衆国 1 0 5 0 4、ニューヨーク州アーモンク (番地なし)

【氏名又は名称】 インターナショナル・ビジネス・マシーンズ・コーポレーション

【代理人】

【識別番号】 100086243

【弁理士】

【氏名又は名称】 坂口 博

【連絡先】 0 4 6 2 - 7 3 - 3 3 1 8、3 3 2 5、3 4 3 1

【選任した代理人】

【識別番号】 100091568

【弁理士】

【氏名又は名称】 市位 嘉宏

【手数料の表示】

【予納台帳番号】 024154

【納付金額】 21,000円

【提出物件の目録】

【物件名】 明細書 1

【物件名】 図面 1

【物件名】 要約書 1

【包括委任状番号】 9304391

【包括委任状番号】 9304392

【ブルーフの要否】 要

【書類名】 明細書

【発明の名称】 プログラム実行方法

【特許請求の範囲】

【請求項 1】

実行途中のメソッドから、コンパイル処理とインタプリタ処理との間の複数の遷移点を含むコードへに実行切り替えを行うプログラム実行方法であって、

遷移点からループ入り口に遷移点を移動させても実行上の問題がない場合には、前記コードの遷移点をループの入り口に移動させるステップと、

遷移点がループの内部に位置する場合には、ループの入り口と遷移点とをポストドミネートする点をループの直前に複製するステップと、

前記コードの移動、プライベートイゼーションおよびコモンサブエクスプレッションエリミネーションが遷移点を超えて行われる場合、超えた遷移点に対してそれらの再計算コードを生成するための情報を遷移点に持たせるステップと、

遷移処理において再計算を実施するステップと
を含むプログラム実行方法。

【請求項 2】

前記メソッドにおいて、インタプリタ処理とコンパイル処理との間で遷移した方が、遷移しない場合に比べて実行が高速化する点を新たな遷移点にするステップをさらに含む

請求項 1 に記載のプログラム実行方法。

【請求項 3】

前記遷移点それぞれにおいて、インタプリタ処理とコンパイル処理との間で遷移するために必要な情報を生成するステップと、

前記生成した情報を前記遷移点それぞれに対応づけて記憶するステップと
をさらに含み、

前記再計算を実施するステップにおいて、前記遷移点それぞれにおいて、前記対応づけられて記憶された情報を用いて再計算を行う

請求項 1 または 2 に記載のプログラム実行方法。

【請求項 4】

実行途中のメソッドを、コンパイル処理とインタプリタ処理との間の複数の遷移点を含むコードへに実行切り替えを行って実行するプログラムであって、

遷移点からループ入り口に遷移点を移動させても実行上の問題がない場合には、前記コードの遷移点をループの入り口に移動させるステップと、

遷移点がループの内部に位置する場合には、ループの入り口と遷移点とをポストドミネートする点をループの直前に複製するステップと、

前記コードの移動、プライベートイゼーションおよびコモンサブエクスプレッションエリミネーションが遷移点を超えて行われる場合、超えた遷移点に対してそれらの再計算コードを生成するための情報を遷移点に持たせるステップと、

遷移処理において再計算を実施するステップと

を含むプログラムを媒介する媒体。

【発明の詳細な説明】

【0001】

【産業上の利用分野】

本発明は、J a v a等の言語で作成されたプログラムを動的にコンパイルするプログラム実行方法に関する。

【0002】

【従来の技術】

近年のJ a v aの普及により動的コンパイラを備える環境の重要性が増している。動的コンパイラによる実行環境では、初期実行時に全てのメソッドを時間のかかる最適化コンパイルしてしまうと、コンパイル時間により逆に速度低下を引き起こすプログラムも存在する。このため最初の実行はインタプリタ実行や最適化を行わない高速コンパイルされた非最適化コードを利用した実行を行い、メソッド単位で起動回数情報やメソッドに多数回繰り返されるループが存在するなどの情報を収集し、それらの情報をもとに最適化コンパイルを実施しすることで全体として高速実行を実現している。

【0003】

例えば、従来から「Adaptive Compilation [HCU91]」と呼ばれる方法が知られている。

この方法は、メソッドの最初の実行時にはほとんど最適化しないでコンパイルを行い、各メソッド毎に `invocation counter` を使って起動回数をカウントし、値が閾値を超えたメソッドに対して最適化レベルを上げて再コンパイルを実施する方法である。

本手法では再コンパイルされたメソッドが実行されるのはそのメソッドを呼び出すタイミングのみのため、1で述べた1回しか起動されないが非常に多数回繰り返される時間のかかるループ（最適化対象とするループであり、多重ループを含む）が存在するメソッドに対して効果がない問題がある。

[HCU91] Urs Hoelzle, Craig Chambers, and David Ungar. Optimizaing Dynamically-Typed Object-Oriented Programming Languages with Polymorphic Inline Caches. In EC00P'91 Conference Proceedings, pp.21-38, Geneva, Switzerland, July 1991. Published as Springer Verlag LNCS 512, 1991.

【0004】

また、従来から「Deoptimization [HCU92]」と呼ばれる方法が知られている。この方法は、最適化したメソッドのコードをデバッグするために最適化されていないコードを作成し、そのコードに実行を遷移させてデバッグするための方法である。この方法では、（1）最適化コンパイルする際にメソッドの先頭とループの後方分岐命令の2種類の位置に `interrupt point` を設定し、（2）`interrupt point` を超えた最適化を行わない制約を加えた最適化コンパイルを行う。（3）実行中に `interrupt` がかった場合は `interrupt point` 単位でプログラムを `resume` し、（4）`interrupt point` のスタックフレームを保存する。（5）メソッドを最適化せずにコンパイルし、（6）スタックフレームを最適化されない状態に合わせて再現して `inturrupt` 状態に入る。本手法では、実行途中での遷移という点で本発明と類似するが、遷移対象が非最適化コードであり遷移点でのレジスタイメージなどの実行状態がソースコードから一意に決定できるため、本手法は遷移先の状態が複雑化する最適化コードへの遷移にはそのままでは適用できない問題がある。

[HCU92] Urs Hoelzle, Craig Chambers, and David Ungar. Debugging Optimized Code With Dynamic Deoptimization. In Proceedings of the SIGPLAN '92

Conference on Programming Language Design and Implementation, pp.21-38
 , June 1992.

【 0 0 0 5 】

また、従来から「Dynamic recompilation [HU94]」と呼ばれる方法が知られている。

この方法は、最適化されていないメソッドの実行途中から最適化コンパイルされたメソッドに、メソッドの実行途中で遷移する手法の概要を示している。この方法では、（１）再コンパイルするメソッドを検出し、（２）実行を再開する点に印を設定し、（３）その点での全ての有効なレジスタが指している値を計算する。これらが成功した場合は（４）スタック上の最適化されていないメソッドを最適化されたメソッドに置き換える。置き換えできない場合は、生成されたコードは後続するそのメソッドの呼び出しで使用されることとなる。本手法はその詳細な処理が記述されていないが、上述した「Deoptimization [HCU92]」の逆処理であると述べられていることから、問題点は以下の通りとなる。

【 0 0 0 6 】

まず本手法では複数の遷移点を扱っていない。また遷移点を越えた最適化を行っていない点、メモリの増加を考慮していない点である。

[HU94] Urs Hoelzle and David Ungar. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. In Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation, pp.326-336. Published as SIGPLAN Notices 29(6), June 1994.

【 0 0 0 7 】

最適化コンパイルを実施した際の実行の切り替え方法は、次のメソッド呼び出しで切りかえる方法と、メソッドの実行途中から最適化コードの実行に遷移する方法の２方法がある。特に後者の方法は、１回しか起動されないが非常に多数回繰り返される時間のかかるループが存在するメソッドの処理を高速化するためには必須の機能である。しかしながらこの実行途中での切り替えに対する従来手法では、以下の２つのような問題が存在した。

【 0 0 0 8 】

【問題 1】 切り替えを行うプログラム上の位置（以下遷移点と呼ぶ）を超えて最適化を実施していない。このため遷移点が存在すると、遷移点が存在しない場合に生成されたコードと比べてコードの質が低下する。後続する当該メソッドの呼び出しによる実行は、遷移点がない状態で最適化コンパイルされた場合に比べて実行速度が低下してしまう問題が生じる。

特に遷移点がループ中に存在する場合は、実行性能に大きく貢献するループ最適化の実施ができなくなるため性能低下が著しくなる。

【0009】

【問題 2】 マルチ・スレッド環境において複数の異なる遷移点から遷移する場合を想定しておらず、遷移が発生する度にコンパイルを実施している。このため複数のコンパイル時間が実行速度の低下を引き起こすとともに、それらのコードが重複して存在してメモリ使用量が増大する問題が生じる。

【0010】

【発明が解決しようとする課題】

本発明は、上述した従来技術の問題点に鑑みてなされたものであり、切り替えを行うプログラム上に遷移点がある場合であっても、遷移点を越えたより高度な最適化が可能なプログラム実行方法を提供することを目的とする。

本発明は、マルチスレッド環境において複数の異なる遷移点から遷移する場合であっても、複数のコンパイルが不要で、しかも、発生したコードに重複を生じさせないプログラム実行方法を提供することを目的とする。

【0011】

【課題を達成するための手段】

上記目的を達成するために、本発明に係るプログラム実行方法は、実行途中のメソッドから、コンパイル処理とインタプリタ処理との間の複数の遷移点を含むコードへに実行切り替えを行うプログラム実行方法であって、遷移点からループ入り口に遷移点を移動させても実行上の問題がない場合には、前記コードの遷移点をループの入り口に移動させるステップと、遷移点がループの内部に位置する場合には、ループの入り口と遷移点とをポストドミネートする点をループの直前に複製するステップと、前記コードの移動、プライベートイゼーションおよびコ

モンサブエクスプレッションエリミネーションが遷移点を超えて行われる場合、超えた遷移点に対してそれらの再計算コードを生成するための情報を遷移点に持たせるステップと、遷移処理において再計算を実施するステップとを含む。

【 0 0 1 2 】

好適には、前記メソッドにおいて、インタプリタ処理とコンパイル処理との間で遷移した方が、遷移しない場合に比べて実行が高速化する点を新たな遷移点にするステップをさらに含む。

【 0 0 1 3 】

好適には、前記遷移点それぞれにおいて、インタプリタ処理とコンパイル処理との間で遷移するために必要な情報を生成するステップと、前記生成した情報を前記遷移点それぞれに対応づけて記憶するステップとをさらに含み、前記再計算を実施するステップにおいて、前記遷移点それぞれにおいて、前記対応づけられて記憶された情報を用いて再計算を行う。

【 0 0 1 4 】

また、本発明に係る媒体は、実行途中のメソッドを、コンパイル処理とインタプリタ処理との間の複数の遷移点を含むコードへに実行切り替えを行って実行するプログラムであって、遷移点からループ入り口に遷移点を移動させても実行上の問題がない場合には、前記コードの遷移点をループの入り口に移動させるステップと、遷移点がループの内部に位置する場合には、ループの入り口と遷移点とをポストドミネートする点をループの直前に複製するステップと、前記コードの移動、プライベートイゼーションおよびコモンサブエクスプレッションエリミネーションが遷移点を超えて行われる場合、超えた遷移点に対してそれらの再計算コードを生成するための情報を遷移点に持たせるステップと、遷移処理において再計算を実施するステップとを含むプログラムを媒介する。

【 0 0 1 5 】

【発明の実施の形態】

【本発明の要点】

まず、本発明の要点を説明する。

【 0 0 1 6 】

[遷移点の存在によってコードの質が低下させないための方法]

遷移点でコードの質が低下する原因は、2つ存在する。1つはループ中に遷移点が存在すると多くのループ最適化処理を施せなくなる点である。またもう1つは、コードの移動およびprivatizationやcommon subexpression eliminationのように、計算途中の状態をレジスタやメモリのローカル領域に保持しておく最適化手法が実施できなくなる点である。これらの問題に対して次のような方法を用いる。

【0017】

(1) 遷移点からループ入り口に遷移点を移動させても実行上の問題がない場合は、コンパイルコードにおける遷移点をループの入り口に移動する。

(2) 遷移点がループ内部に位置する場合、遷移点の存在がループ最適化に影響を与えないようにするために、ループの入り口から、ループの入り口と遷移点とともにpost-dominateする点までをループ入り口の直前に複製し、遷移点がループの外側もしくはループの入り口と等しくなるようにループを変形する。

(3) コードの移動およびprivatizationやcommon subexpression eliminationが遷移点を超えて行われる場合、超えた遷移点に対してそれらの再計算コードを生成するための情報を遷移点に持たせておき、遷移処理において再計算を実施する。

【0018】

[1つのコードで通常の呼び出しと途中からの遷移に対応する方法]

インタプリタからこれから遷移しようとする点とともに、メソッドのプログラムを検索して遷移する可能性がありかつ遷移により高速化の効果のある点を検出し、これらの点も遷移点とする。

各遷移点に対して、その遷移点でのレジスタ使用状態、再計算コード情報からなる遷移情報を生成し、元のコード上のアドレスとあわせて、これらを要素とするテーブルを生成し、メソッドからアクセスできる場所に設定する。遷移実行時には、遷移点のアドレスをキーとしてこのテーブルをアクセスし、遷移情報を取得する。

【0019】

インタプリタが使用しているスタックフレームからコンパイルされたコードで使用するスタックフレームへの切替時に不要な変数領域を生成しない方法。インタプリタでは各メソッドで指定されたローカル変数の数全てがスタックフレーム中に割り当てられる。一方コンパイルされたコードでは、コンパイルされたコードでは、複数のローカル変数が同一の退避場所を使用していたり、レジスタのみに値が保持されスタックフレーム上へは退避されない変数が存在したりする。このためメソッドで指定したローカル変数全てがスタックフレーム内に割り当てられるわけではない。この問題に対して次の方法を用いる。

【 0 0 2 0 】

コンパイルされたコードが必要とするスタックフレーム中のローカル変数領域の大きさを、コンパイル時にメソッドからアクセスできる場所に格納しておく。このとき領域の大きさは、コンパイルされたコードが必要とする分のみを割り当てる。また、遷移情報には、遷移点で生きている変数に対してそれがレジスタに保持されているか、スタックフレーム中に保持されているかの情報を持たせる。遷移時には、まず、インタプリタが使用しているスタックフレームを退避し、コンパイルされたコードのためのスタックフレームを生成した後、遷移情報を使用して必要なローカル変数の値をレジスタ上かスタックフレーム中にコピーする。最後にインタプリタのスタックフレームを除去して、遷移点に実行を移す。

インタプリタから機械コード実行への切り替えが、動的コンパイラを起動した最初の 1 回の場合に限るときは、緩衝コードをスタック領域に生成することで緩衝コードに対するメモリ使用の増加をなくすることができる。

【 0 0 2 1 】

以下、本発明の手順を説明する。

〔 3. 1. インタプリタが途中遷移を決定した場合 〕

インタプリタが途中遷移を決定した場合、以下の処理を行う。

(3. 1. 1) 既にメソッドがコンパイルされている場合は、メソッドより遷移情報テーブルを取得し、遷移点の命令のアドレスから遷移情報の取得を試みる。遷移情報を取得できた場合は、 3. 3 の 2 以降の処理を行う。遷移情報を取得できなかった場合は、この点での遷移処理を中止しインタプリタによる実行を継続

する。

【0022】

(3. 2. 1) 動的コンパイラを起動する。このとき遷移を決定した命令のアドレスと遷移後に実行される命令のアドレスをコンパイラに渡す。遷移後に実行される命令のアドレスが自明な場合にはこれを省略することもできる。

【0023】

[3. 2 動的コンパイラの処理]

動的コンパイラで以下の処理を行う。

(3. 2. 1) メソッドのコードを、Basic Block (以下BBと省略する) に分割し、Control Flow Graphを生成する。このとき、インタプリタが指定した遷移点がBasic Blockの先頭となるようにする。

【0024】

(3. 2. 2) インタプリタが指定した遷移点を含むBBに印を付ける。

【0025】

(3. 2. 3) メソッドのコードで、インタプリタが指定した遷移点以外に遷移される可能性のあり、かつ遷移による効果がある遷移候補点を探索する。この処理はインタプリタが指定した遷移点以外を遷移点としない場合は実施しない。遷移候補点はメソッド中のインタプリタが検出する全位置から遷移効果がないとコンパイラが判断可能な位置を除いたものとする。遷移効果がない点とは、ループ外の点などである。

【0026】

(3. 2. 4) 遷移点からループ入り口に遷移点を移動させても実行上の問題がない場合は、コンパイルコードにおける遷移点をループの入り口に移動する。移動上の問題がある場合とは、ヒープ領域への書き込み命令が存在する場合、およびループの入り口から遷移点の間にある変数が定義されており、ループの入り口からその定義点までの間にその変数の参照が存在し、かつループの入り口においてその変数の値を再計算することができない場合である。再計算可能な場合は、遷移点に対する再計算情報を生成する。

【0027】

(3. 2. 5) 遷移点がループ内部に位置する場合。

遷移点がループ内部に位置する場合、遷移点の存在がループ最適化に影響を与えないようにするために、遷移点がループ（最適化対象とするループであり、多重ループを含む）の開始点となるようにループを変形する。変形方法は、ループのentry pointから、ループのentry pointと遷移点をともにpost-dominateするループ内の点までのコードを複製する操作を行う。ここで複製されるコードは最悪の場合、そのループのコードよりも小さい。さらにインタプリタから遷移される点がループの後方分岐命令に限定される場合は、J a v a のforループではループの制御コードのみを複製すればよく、do-whileループではまったく複製する必要がない。

【0028】

(3. 2. 6) コードの移動やcommon subexpressionとprivatizationによる値のキャッシングが遷移点を超えて行われる場合、超えた遷移点に対してそれらの再計算コードを生成するための情報をそれらが超えた全ての遷移点に登録する。

【0029】

(3. 2. 7) メソッドに対するコードを生成する。

【0030】

(3. 2. 8) コンパイルされたコードで使用するローカル変数領域の大きさをメソッドからアクセスできる場所に格納する。

【0031】

(3. 2. 9) 各遷移点に対して、遷移点のコードアドレス、キャッシュの再計算コード情報、および遷移点のレジスタ使用情報からなる遷移情報を生成し、元のコード上の遷移命令のアドレスとついにした遷移情報テーブルを作成する。このテーブルをメソッドからアクセスできる場所に格納する。遷移情報は、実際のコードとしてあらわしてもよいし、共通の遷移ルーチンに渡すデータとして表してもよい。またインタプリタから機械コード実行への切り替えが動的コンパイラを起動した最初の1回の場合に限るときは、遷移情報をスタック領域に生成することで、遷移が終わったときに即時除去してしまうことも可能である。

【0032】

[3 . 3 . インタプリタの処理]

インタプリタで以下の処理を行う。

(3 . 3 . 1) 動的コンパイルの結果よりコンパイルが成功した場合は、遷移情報テーブルより遷移情報を取得する。コンパイルが失敗した場合は、この点での遷移処理を中止しインタプリタによる実行を継続する。

[0 0 3 3]

(3 . 3 . 2) コンパイルされたコードで使用するローカル変数領域の大きさを獲得し、インタプリタで使用していたスタックフレーム中の情報のうち必要なものを退避してからコンパイルされたコード用のスタックフレームを生成する。

[0 0 3 4]

(3 . 3 . 3) 遷移情報を使用してコンパイルされたコード用のスタックフレームに必要な情報を設定する。遷移情報がコードで表される場合はコードを実行する。遷移情報がデータとして表される場合は、遷移処理ルーチンにこのデータを渡して遷移処理を実施する。遷移処理の手順としては、(1) スタック領域に格納される値を退避領域からコピーする、(2) 再計算情報により値の再計算を行い、結果をローカル変数かレジスタに設定するか退避領域に設定する、(3) レジスタに値をロードする、などにより実現できる。

[0 0 3 5]

(3 . 3 . 4) 遷移点のアドレスより実行を開始する。

[0 0 3 6]

[プログラム実行装置 1]

図 1 は、本発明に係るプログラム実行装置 1 の構成を示す図である。

図 1 に示すように、プログラム実行装置 1 は、サーバ 1 0 とクライアント 1 2 とが、ネットワーク 1 0 6 を介して接続されて構成される。

サーバ 1 0 は、J a v a ソースコード 1 0 0、J a v a バイトコードコンパイラ (J A V A C) 1 0 2 および J a v a バイトコード 1 0 4 を含む。

[0 0 3 7]

クライアント 1 2 は、記録媒体 1 4 0 を介してクライアント 1 2 に供給され、記憶装置 1 4 を介してハードウェア 1 3 2 で実行される J a v a 処理プログラム

120およびOS130を含む。

Java処理プログラム120は、Javaバイトコードベリファイヤ122、Javaインタプリタ124、JITコンパイラ126およびネイティブコードエグゼキューション128を含む。

【0038】

サーバ10において、Javaソースコード100は、JAVAC102などのバイトコードコンパイラによりバイトコード104に変換され、ネットワーク106を介してクライアント12に供給される。

【0039】

クライアント12において、バイトコードベリファイヤ122は、サーバ10からのバイトコード104を検証する。

【0040】

Javaインタプリタ124は、JITコンパイラ126が、検証されたバイトコード104をコンパイルしない場合に、検証されたバイトコード104を実行する。

【0041】

JITコンパイラは、検証されたバイトコード104をコンパイルする場合には、バイトコード104をコンパイル処理してネイティブコードを生成する。

【0042】

ネイティブコードエグゼキューション128は、コンパイル処理の結果として生成されたネイティブコードを実行する。

【0043】

ここでは、Javaインタプリタ124とJITコンパイラ126への組み込む場合に対する実施例を示す。

実施例で使用する条件は以下の通りである。

Javaインタプリタ124はループ（最適化対象とするループであり、多重ループを含む）の後方分岐となる命令でのみ遷移を行う。遷移を行わないことに決めた命令にはフラグを設定し、後続するThreadで遷移しないようにする。

遷移後は、遷移を決定した後方分岐命令の分岐先命令から実行を開始するもの

とする。

【 0 0 4 4 】

J a v a インタプリタ 1 2 4 では遷移点を、遷移先のアドレスではなく遷移を決定した後方分岐命令で行う。

J I T コンパイラ 1 2 6 では、J I T コンパイラ 1 2 6 を起動した J a v a インタプリタ 1 2 4 が遷移を決定した後方分岐命令に加えて、他の後方分岐命令のうちでフラグが設定されておらずかつ最適化対象となった

ループの後方分岐命令を遷移点として検出する。

遷移点をループ外へ移動するために複製されるコードの量を少なくするために、遷移点をより外のループは最適化対象としない。この制限は、次の点で妥当である。まず外側のループが for ループの場合、J a v a インタプリタ 1 2 4 によりループの条件判定を 1 度以上実施しているので、その段階で遷移しないことから繰り返し回数が少ないループと考えられる。

遷移情報は実行コードとして生成する。

J I T コンパイラ 1 2 6 で生成されるコードでは、元のプログラムに含まれるローカル変数領域は必ずスタックフレーム上に割り当てられるものとする（説明を簡単にするため）。

【 0 0 4 5 】

なお、実施例中では以下の用語を使用する。

transfer point ... J a v a インタプリタ 1 2 4 から J I T コンパイラ 1 2 6 に遷移する点

transfer bwd jump ... J a v a インタプリタ 1 2 4 で transfer point を検出した後方分岐命令

transfer bwd jump address ... 実行中のメソッドの bytecode 中の transfer bwd jump 命令が格納されているメモリのアドレス

transfer point table ... transfer bwd jump address と遷移のための pad code の entry address の組を要素とするテーブル

メソッド情報（以下 mb と略す） ... J a v a でメソッドの情報をもつ構造体

メソッドコンパイル情報（以下minfoと略す） ... コンパイル時にメソッドに関するさまざまな情報を蓄積する構造体。

【 0 0 4 6 】

[J a v a インタプリタ 1 2 4 の処理（処理 1）]

J a v a インタプリタ 1 2 4 で、 J I T コンパイラ 1 2 6 に即時遷移する backedgeを検出する。

必要があればレジスタにキャッシュしている値をメモリに書き出す。

J I T コンパイラ 1 2 6 を呼んでコンパイルを実行する。このとき、遷移のきっかけとなったbackedgeのアドレスを J I T コンパイラ 1 2 6 に渡す。

【 0 0 4 7 】

[J I T コンパイラ 1 2 6 の処理（処理 2）]

コンパイルを排他処理するためのロックをかける。

transfer bwd jump addressが渡された場合で既にコンパイル済の場合、遷移点テーブルを検索し、該当する遷移点アドレスがテーブル上に存在する場合、そのテーブル・オフセットを戻り値として返す（終了）。

transfer bwd jump addressが渡された場合、メソッド情報に遷移点に対する処理を実施することを表すフラグ（以下このフラグをgen_tpフラグと呼ぶ）を設定する。

ループの後方分岐の数をカウントし、transfer point tableをアロケートする。

Bytecodeをtraverseし、Basic Block（以下B Bと呼ぶ）およびループを構築する。この際、3のフラグが設定されている場合には、以下の処理を行う。

【 0 0 4 8 】

J a v a インタプリタ 1 2 4 から受け取ったtransfer bwd jumpから得たtransfer pointのB Bをメソッド情報に登録する。

B Bに遷移点をあらわすフラグを立てる。このフラグが立っているB Bは前者B Bなどとマージしては行けない。

ループを形成する後方分岐で、遷移しないことを示すフラグがたっていない命令をtransfer point tableに登録する。

【 0 0 4 9 】

ループにTransfer pointが含まれる場合は、以下の処理を行う。

Control flow graphに従って、ループの入り口から遷移点までのコードをトレースする。トレース時には、次の命令の存在をチェックする。これらの命令が存在しない場合はTransfer pointをループ入り口まで移動する。

【 0 0 5 0 】

つまり、Control flow graphにしたがって、ループの入り口から遷移点までのコードをトレースする。

次に、トレース時に、次の命令の存在をチェックする。これらの命令が存在しない場合は、Transfer pointをループ入り口に変更し、ループ外への移動処理を終了するヒープ領域への書き込み命令を実行する。

次に、ローカル変数の定義命令を実行する。

さらに、ループ入り口より遷移店に至るコードをループ外に複製し、遷移点をループ外へ移動する。

【 0 0 5 1 】

common subexpression eliminationを実施する。

Transfer pointをまたがる場合、除去される計算がJ a v a インタプリタ 1 2 4 がもつローカル変数から再計算できる場合は、再計算情報としてそのTransfer pointに登録する。再計算できない場合は、そのTransfer pointをこえて処理を実施しないように制御する。

【 0 0 5 2 】

field変数に対するprivatizationを実施する。

Transfer pointをまたがる場合、除去される計算がJ a v a インタプリタ 1 2 4 がもつローカル変数から再計算できる場合は、再計算情報としてそのTransfer pointに登録する。再計算できない場合は、そのTransfer pointをこえて処理を実施しないように制御する。

【 0 0 5 3 】

コード生成では以下の処理を行う。

minfoのtransfer point tableの情報を使って、mbからアクセスできる場所に

、transfer pointテーブル{(元のコードのアドレス, 遷移コード)を要素とする配列}と、テーブルサイズ格納領域を作成し、transfer pointのbytecode addressesを設定する。

【0054】

このとき、検索を容易にするためbytecode addressを小さい順に並べ替えて生成する。

transfer point tableの情報とtransfer pointのレジスタイメージを用いて、キャッシュの再計算コードと各transfer pointのための遷移コード(メモリ上の値をレジスタに載せるコード)を生成し、transfer pointテーブルにtransfer pointコードアドレスとして設定する。

JITコンパイラ126が必要とする作業領域サイズ(ローカル変数領域とスタック領域)をmbからアクセスできる場所に格納する。

【0055】

[Javaインタプリタ124側の処理(処理3)]

コンパイラの戻り値をチェックする。コンパイルに失敗していたら、Javaインタプリタ124実行を継続する。(終了)

mbのtransfer point tableを取得し、現在のbwd jump addressがtable上にあるかどうかを調べる。table上にaddressがなければ、Javaインタプリタ124実行を継続する。(終了)

JITコンパイラ126から受け取ったワークエリアサイズを加味して、スタックフレームをJavaインタプリタ124用のものからJITコンパイラ126用のものに変換する。

【0056】

mbからtransfer pointテーブルを獲得する。

transfer pointテーブルと、現在実行しているbytecode addressからtransfer pointコードアドレスを獲得する。

transfer pointコードアドレスより実行を開始する。

【0057】

以下、さらに図2～図8に示すフローチャートを参照して、本発明に係るプロ

グラム実行装置 1 の処理を説明する。

本発明は、バイトコード実行処理（図 2 ; S 1 0）において、S 1 4 の処理として位置づけられる。

図 2 に示すように、ステップ 1 0 0（S 1 0 0）において、プログラム実行装置 1 は、メソッドを実行する。

ステップ 1 0 2（S 1 0 2）において、プログラム実行装置 1 は、メソッドをインタプリタで実行するか否かを判断し、インタプリタで実行する場合には S 1 0 4 の処理に進み、これ以外の場合には S 1 1 4 の処理に進む。

【 0 0 5 8】

ステップ 1 0 4（S 1 0 4）において、プログラム実行装置 1 は、インタプリタでの処理を開始する。

【 0 0 5 9】

ステップ 1 0 6（S 1 0 6）において、プログラム実行装置 1 は、命令を読み込む。

【 0 0 6 0】

ステップ 1 0 8（S 1 0 8）において、プログラム実行装置 1 は、コンパイルされたコードを実行するか否かを判断し、実行する場合には S 1 4 の S 1 6 の処理に進み、これ以外の場合には S 1 1 0 の処理に進む。

【 0 0 6 1】

ステップ 1 1 0（S 1 1 0）において、プログラム実行装置 1 は、コードをインタプリタで実行する。

【 0 0 6 2】

ステップ 1 1 2（S 1 1 2）において、プログラム実行装置 1 は、インタプリタでの処理を終了する。

【 0 0 6 3】

ステップ 1 1 4（S 1 1 4）において、プログラム実行装置 1 は、コンパイラによりメソッドをコンパイルする。

【 0 0 6 4】

ステップ 1 1 6（S 1 1 6）において、プログラム実行装置 1 は、コンパイル

により生成されたネイティブを実行する。

【0065】

図3は、図2に示した本発明に係るS14の処理を示す図である。

図3に示すように、S14内の遷移のための前処理（S16）のステップ160（S160）において、プログラム実行装置1は、メソッドがコンパイル済みか否かを判断し、コンパイル済みの場合にはS162の処理に進み、これ以外の場合にはS20（図4～図8を参照して後述）の処理に進む。

【0066】

ステップ162（S162）において、プログラム実行装置1は、メソッドより遷移情報テーブルを取得する。

【0067】

ステップ164（S164）において、プログラム実行装置1は、遷移点の命令アドレスをキーとして遷移情報テーブルを検索する。

【0068】

ステップ142（S142）において、プログラム実行装置1は、遷移情報が存在するか否かを判断し、存在する場合には、遷移処理（S18）に進み、これ以外の場合にはS146の処理に進む。

【0069】

ステップ144（S144）において、プログラム実行装置1は、S20の処理においてコンパイルが成功したか否かを判断し、成功した場合にはS18の処理に進み、これ以外の場合にはS146の処理に進む。

【0070】

ステップ146（S146）において、プログラム実行装置1は、インタプリタでの実行処理に戻る。

【0071】

ステップ148（S148）において、プログラム実行装置1は処理を終了する。

【0072】

遷移処理（S18）のステップ180（S180）において、プログラム実行

装置 1 は、遷移情報テーブルから遷移情報を取得する。

【 0 0 7 3 】

ステップ 1 8 2 (S 1 8 2) において、プログラム実行装置 1 は、現在のスタックフレーム内の必要情報を退避する。

【 0 0 7 4 】

ステップ 1 8 4 (S 1 8 4) において、プログラム実行装置 1 は、遷移情報を用いてスタックフレームのサイズを変更する。

【 0 0 7 5 】

ステップ 1 8 6 (S 1 8 6) において、プログラム実行装置 1 は、遷移情報と退避情報からネイティブコード用のスタックフレームを生成する。

【 0 0 7 6 】

ステップ 1 8 8 (S 1 8 8) において、プログラム実行装置 1 は、遷移情報より遷移アドレスを取得し、そこより実行を開始する。実行完了後、プログラム実行装置 1 は、処理を終了する (S 1 4 8) 。

【 0 0 7 7 】

図 4 は、図 3 に示した遷移のためのコンパイル処理 (S 2 0) を示すフローチャートである。

S 2 0 のステップ 2 0 2 (S 2 0 0) において、プログラム実行装置 1 は、メソッドのコードをベーシックブロック (Basic Block) に分割する。

【 0 0 7 8 】

ステップ 2 0 4 (S 2 0 4) において、プログラム実行装置 1 は、遷移点がベーシックブロックの先頭か否かを判断し、先頭の場合には S 2 0 8 の処理に進み、これ以外の場合には S 2 0 6 の処理に進む。

【 0 0 7 9 】

ステップ 2 0 6 (S 2 0 6) において、プログラム実行装置 1 は、遷移点がベーシックブロックの先頭になるようにベーシックブロックを分割する。

【 0 0 8 0 】

ステップ 2 0 8 (S 2 0 8) において、プログラム実行装置 1 は、制御フローグラフの生成を行う。

ステップ 2 1 0 (S 2 1 0) において、プログラム実行装置 1 は、遷移点のベーシックブロックに印を設定し、他の遷移可能点の検索処理 (S 2 2 ; 図 5) の処理に進む。

【 0 0 8 1 】

図 5 は、図 4 に示した他の遷移可能点の検索処理 (S 2 2) を示すフローチャートである。

図 5 に示すように、S 2 2 のステップ 2 2 2 (S 2 2 2) において、プログラム実行装置 1 は、メソッド内の遷移点以外のコードに対して繰り返す処理を開始する。

【 0 0 8 2 】

ステップ 2 2 4 (S 2 2 4) において、プログラム実行装置 1 は、インタプリタから遷移する可能性がある命令であるか否かを判断し、可能性がある場合には S 2 2 6 の処理に進み、これ以外の場合には S 2 3 6 の処理に進む。

【 0 0 8 3 】

ステップ 2 2 6 (S 2 2 6) において、プログラム実行装置 1 は、遷移する効果があるかどうかを解析する。

【 0 0 8 4 】

ステップ 2 2 8 (S 2 2 8) において、プログラム実行装置 1 は、遷移効果があるか否かを判断し、ある場合には S 2 3 0 の処理に進み、これ以外の場合には S 2 3 2 の処理に進む。

【 0 0 8 5 】

ステップ 2 3 0 (S 2 3 0) において、プログラム実行装置 1 は、遷移点がベーシックブロックの先頭か否かを判断し、先頭である場合には S 2 3 4 の処理に進み、これ以外の場合には S 2 3 2 の処理に進む。

【 0 0 8 6 】

ステップ 2 3 2 (S 2 3 2) において、プログラム実行装置 1 は、遷移点がベーシックブロックの先頭になるようにベーシックブロックを分割し、制御フローグラフを修正する。

【 0 0 8 7 】

ステップ 2 3 4 (S 2 3 4) において、プログラム実行装置 1 は、遷移点のベーシックブロックに印を設定する。

【0 0 8 8】

ステップ 2 3 6 (S 2 3 6) において、プログラム実行装置 1 は、メソッドの中の遷移点以外のコードに対して繰り返す処理を終了する。

【0 0 8 9】

ステップ 2 3 8 (S 2 3 8) において、プログラム実行装置 1 は、S 2 2 の処理を終了して、ループを変形せずにループ内の遷移点のループ外への移動処理 (S 2 6 ; 図 6) を行う。

【0 0 9 0】

図 6 は、図 4 に示したループを変形せずにループ内の遷移点のループ外への移動処理 (S 2 6) を示すフローチャートである。

図 6 に示すように、S 2 6 のステップ 2 6 2 (S 2 6 2) において、プログラム実行装置 1 は、S 3 0 0 までのループ 1 の処理を、各遷移点ごとに繰り返す。

【0 0 9 1】

ステップ 2 6 4 (S 2 6 4) において、プログラム実行装置 1 は、遷移点が最適化可能なループの中に位置するか否かを判断し、位置する場合には S 2 6 6 の処理に進み、これ以外の場合には S 3 0 0 の処理に進む。

【0 0 9 2】

ステップ 2 6 6 (S 2 6 6) において、プログラム実行装置 1 は、ループ内にメソッド以外からも参照可能な領域への書き込み命令が存在するか否かを判断し、存在する場合には S 3 0 0 の処理に進み、これ以外の場合には S 2 6 8 の処理に進む。

【0 0 9 3】

ステップ 2 6 8 (S 2 6 8) において、プログラム実行装置 1 は、ループ外で定義された値がループ内で参照されるローカル変数の集合 V_r を計算する。

【0 0 9 4】

ステップ 2 7 0 (S 2 7 0) において、プログラム実行装置 1 は、ループ内で定義されるローカル変数の集合 V_d を計算する。

【 0 0 9 5 】

ステップ 2 7 2 (S 2 7 2) において、プログラム実行装置 1 は、S 2 8 2 までのループ 2 の処理を、再計算が必要なローカル変数の集合 $(V_r \cap V_d)$ の各変数 V_i に対して繰り返す。

【 0 0 9 6 】

ステップ 2 7 4 (S 2 7 4) において、プログラム実行装置 1 は、ループ開始点で V_i を再計算するための情報を収集する。

【 0 0 9 7 】

ステップ 2 7 6 (S 2 7 6) において、プログラム実行装置 1 は、ループ開始点において V_i を再計算することが可能か否かを判断し、可能な場合には S 2 7 8 の処理に進み、これ以外の場合には S 3 0 0 の処理に進む。

【 0 0 9 8 】

ステップ 2 7 8 (S 2 7 8) において、プログラム実行装置 1 は、 V_i の再計算に必要なローカル変数の集合 V_c を計算する。

【 0 0 9 9 】

ステップ 2 8 0 (S 2 8 0) において、プログラム実行装置 1 は、集合 $(V_c \cap V_d)$ が空集合であるか否かを判断し、空集合である場合には S 2 8 2 の処理に進み、これ以外の場合には S 3 0 0 の処理に進む。

【 0 1 0 0 】

ステップ 2 8 4 (S 2 8 4) において、プログラム実行装置 1 は、遷移点をループの開始点に移動する。

【 0 1 0 1 】

ステップ 2 8 6 (S 2 8 6) において、プログラム実行装置 1 は、集合 $(V_r \cap V_d)$ が空集合であるか否かを判断し、空集合である場合には S 3 0 0 の処理に進み、これ以外の場合には S 2 8 8 の処理に進む。

【 0 1 0 2 】

ステップ 2 8 8 (S 2 8 8) において、プログラム実行装置 1 は、ベーシックブロックを 1 つ生成する。

【 0 1 0 3 】

ステップ 290 (S290) において、プログラム実行装置 1 は、新たに生成したベーシックブロックからループ開始点への制御フロー上の経路を生成する。

【0104】

ステップ 292 (S292) において、プログラム実行装置 1 は、ループ外からループ開始点への経路を、新たに作成したベーシックブロックへの経路に変更する。

【0105】

ステップ 294 (S294) において、プログラム実行装置 1 は、S298までのループ 3 の処理を、再計算を行う各ローカル変数に対して繰り返す。

【0106】

ステップ 296 (S296) において、プログラム実行装置 1 は、再計算情報を用いて再計算コードを生成し、新たに生成したベーシックブロックのコードに追加する。

【0107】

ステップ 302 (S302) の処理において、プログラム実行装置 1 は、ループ変形による遷移点のループ外への移動処理 (S32 ; 図 7) に進む。

【0108】

図 7 は、図 4 に示したループ変形による遷移点のループ外への移動処理 (S32) を示すフローチャートである。

【0109】

図 7 に示すように、S32 のステップ 322 (S322) において、プログラム実行装置 1 は、S234 までのループ 1 の処理を、各遷移点ごとに繰り返す。

【0110】

ステップ 324 (S324) において、プログラム実行装置 1 は、遷移点が最適化可能なループの中に位置するか否かを判断し、位置する場合には S286 の処理に進み、これ以外の場合には S334 の処理に進む。

【0111】

ステップ 326 (S326) において、プログラム実行装置 1 は、ループ開始点と遷移点をともにイミディエイトリイポストドミネート(immediately post-dom

inate)するループ内のベーシックブロック B B p を求める。

【 0 1 1 2 】

ステップ 3 2 8 (S 3 2 8) において、プログラム実行装置 1 は、ループ開始点から B B p に至るすべての経路の和で構成される制御フローグラフの複製を生成する。

【 0 1 1 3 】

ステップ 3 3 0 (S 3 3 0) において、プログラム実行装置 1 は、ループ外からループ開始点に接続している全ての制御フローを、複製したフローグラフでループ開始点に対応するベーシックブロックへの制御フローに変更する。

【 0 1 1 4 】

ステップ 3 3 2 (S 3 3 2) において、プログラム実行装置 1 は、複製した制御フローグラフから B B p への制御フローはそのまま残す。

【 0 1 1 5 】

ステップ 3 3 6 (S 3 3 6) において、プログラム実行装置 1 は、図 4 に示した S 2 0 の S 2 0 8 の処理に戻る。

【 0 1 1 6 】

再び図 4 を参照する。

ステップ 2 0 8 (S 2 0 8) の処理において、プログラム実行装置 1 は、遷移点を考慮したりダンダンシイエリミネーション (redundancy elimination) を行う。

【 0 1 1 7 】

ステップ 2 1 0 (S 2 1 0) の処理において、プログラム実行装置 1 は、遷移点を考慮する必要のない最適化処理を行う。

【 0 1 1 8 】

ステップ 2 1 2 (S 2 1 2) において、プログラム実行装置 1 は、メソッドに対するネイティブコードの生成を行い、遷移情報の生成処理 (S 3 4 ; 図 8) に進む。

【 0 1 1 9 】

図 8 は、図 4 に示した遷移情報の生成処理 (S 3 4) を示すフローチャートで

ある。

S 3 4 のステップ 3 4 2 (S 3 4 2) において、プログラム実行装置 1 は、コンパイルされたコードで使用するスタックフレームのサイズをメソッドからアクセスできる場所に格納する。

【0 1 2 0】

ステップ 3 4 4 (S 3 4 4) において、プログラム実行装置 1 は、S 3 2 8 までのループ 1 の処理を、各遷移点ごとに繰り返す。

【0 1 2 1】

ステップ 3 4 6 (S 3 4 6) において、プログラム実行装置 1 は、遷移情報 (遷移点のコードアドレス、リダンダンシエリミネーションによるキャッシュの再計算情報、遷移点のレジスタ使用状態情報、インタプリタが遷移を決定した命令のアドレス) を生成する。

【0 1 2 2】

ステップ 3 5 0 (S 3 5 0) において、プログラム実行装置 1 は、図 3 に示した S 1 4 4 (図 3) の処理に戻る。

【0 1 2 3】

[プログラム実行装置 1 の効果]

プログラム実行装置 1 が最適化する「途中から実行を遷移することが性能に大きく貢献するメソッド」は、たとえばループを含むメソッドで、そのループが無限ループであったり、多数回まわり、インタプリタ実行とコンパイルコードの実行で優位に差が出てしまったりするメソッドがあげられる。

実行せずにループを含むかどうかを調べるには実行前にコードを走査する必要がある。しかしながらコードサイズが大きい場合などは走査にかかる時間自体が性能低下の要因となりうる。

【0 1 2 4】

プログラム実行装置 1 では、このような走査をせずに実行しながらループを検出し、コンパイルコードに遷移することで高速な実行を可能とする。

インタプリタからコンパイルコードへの遷移は、一般には任意の点であり、また遷移すべき状況を考慮すると遷移点がループ内に位置する場合がほとんどであ

る。しかしながら、遷移点がループ内に存在したままではそのループを最適化することが困難となることから、遷移点をループ外へ移動する処理は性能向上のために重要な処理となる。

【 0 1 2 5 】

【発明の効果】

以上説明したように、本発明に係るプログラム実行方法は、切り替えを行うプログラム上に遷移点がある場合であっても、遷移点を越えたより高度な最適化が可能である。

また、本発明に係るプログラム実行方法によれば、マルチスレッド環境において複数の異なる遷移点から遷移する場合であっても、複数回のコンパイルが不要で、しかも、発生したコードに重複を生じさせることがない。

【図面の簡単な説明】

【図 1】

本発明に係るプログラム実行装置の構成を示す図である。

【図 2】

本発明に係るプログラム実行装置の処理を示す第 1 のフローチャートである。

【図 3】

本発明に係るプログラム実行装置の処理を示す第 2 のフローチャートである。

【図 4】

本発明に係るプログラム実行装置の処理を示す第 3 のフローチャートである。

【図 5】

本発明に係るプログラム実行装置の処理を示す第 4 のフローチャートである。

【図 6】

本発明に係るプログラム実行装置の処理を示す第 5 のフローチャートである。

【図 7】

本発明に係るプログラム実行装置の処理を示す第 6 のフローチャートである。

【図 8】

本発明に係るプログラム実行装置の処理を示す第 7 のフローチャートである。

【符号の説明】

1 . . . コンパイラ

10 . . . サーバ

100 . . . J a v a ソースコード

102 . . . J a v a バイトコードコンパイラ (JAVAC)

104 . . . J a v a バイトコード

12 . . . クライアント

140 . . . 記録媒体

14 . . . 記憶装置

132 . . . ハードウェア

120 . . . J a v a 処理プログラム

122 . . . J a v a バイトコードベリファイヤ

124 . . . J a v a インタプリタ

126 . . . J I T コンパイラ

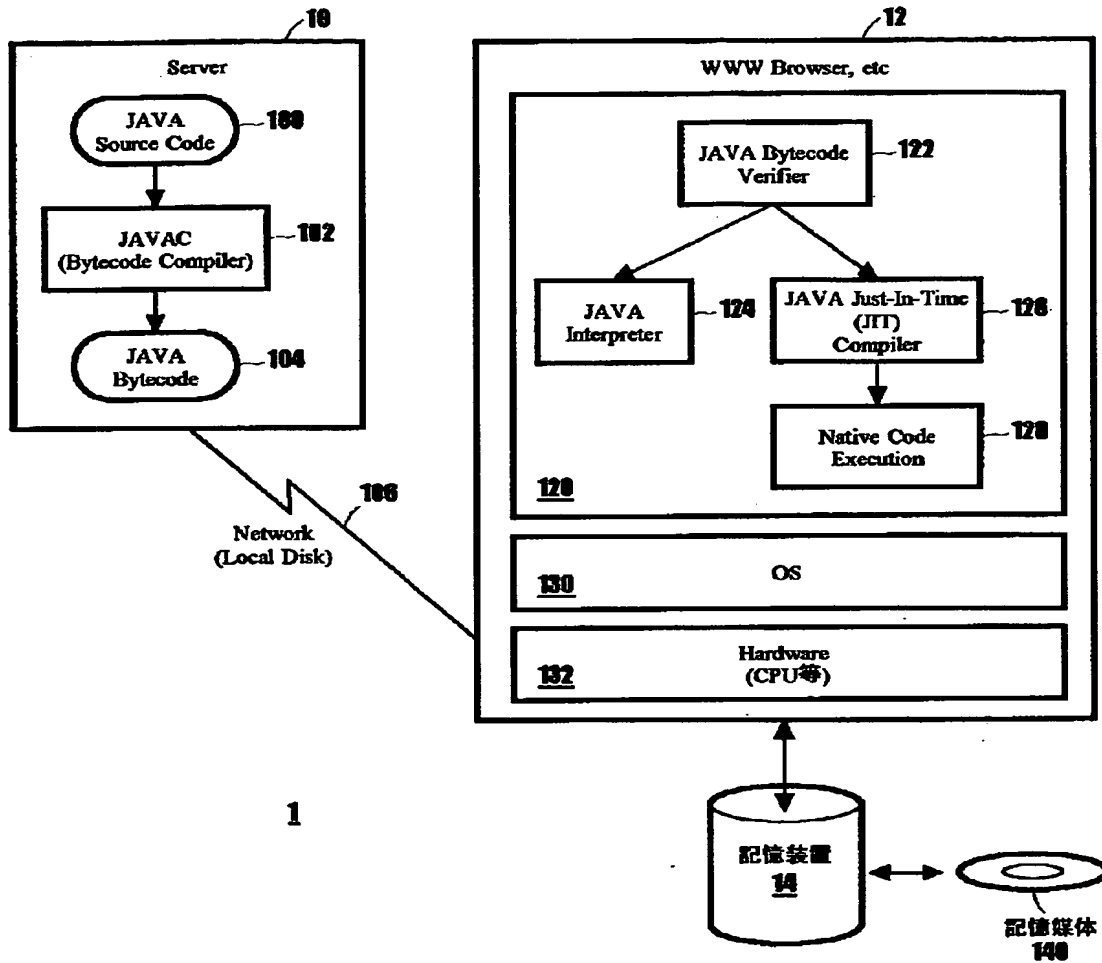
128 . . . ネイティブコードエクスキューション

130 . . . O S 130

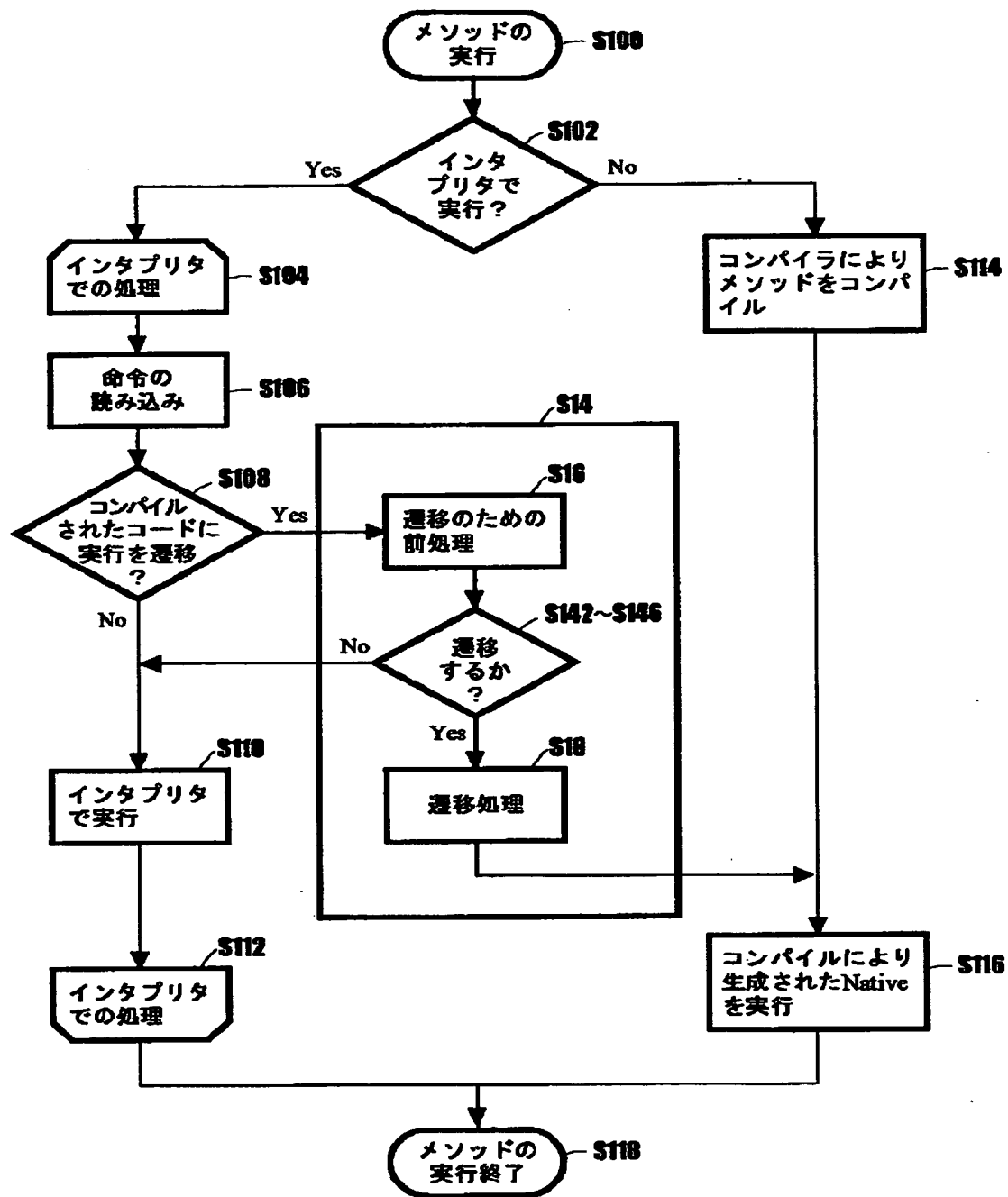
106 . . . ネットワーク

【書類名】 図面

【図 1】

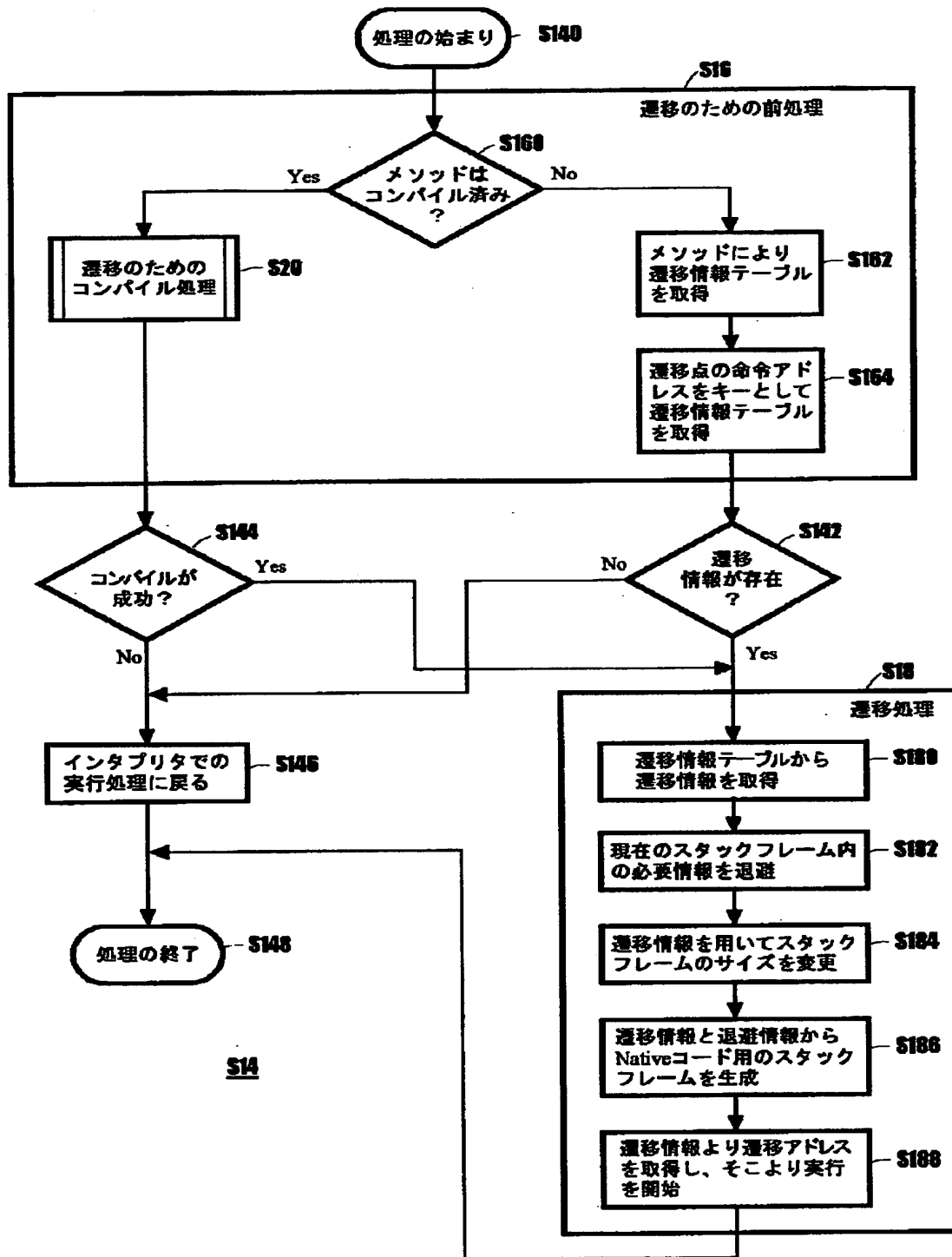


【図 2】

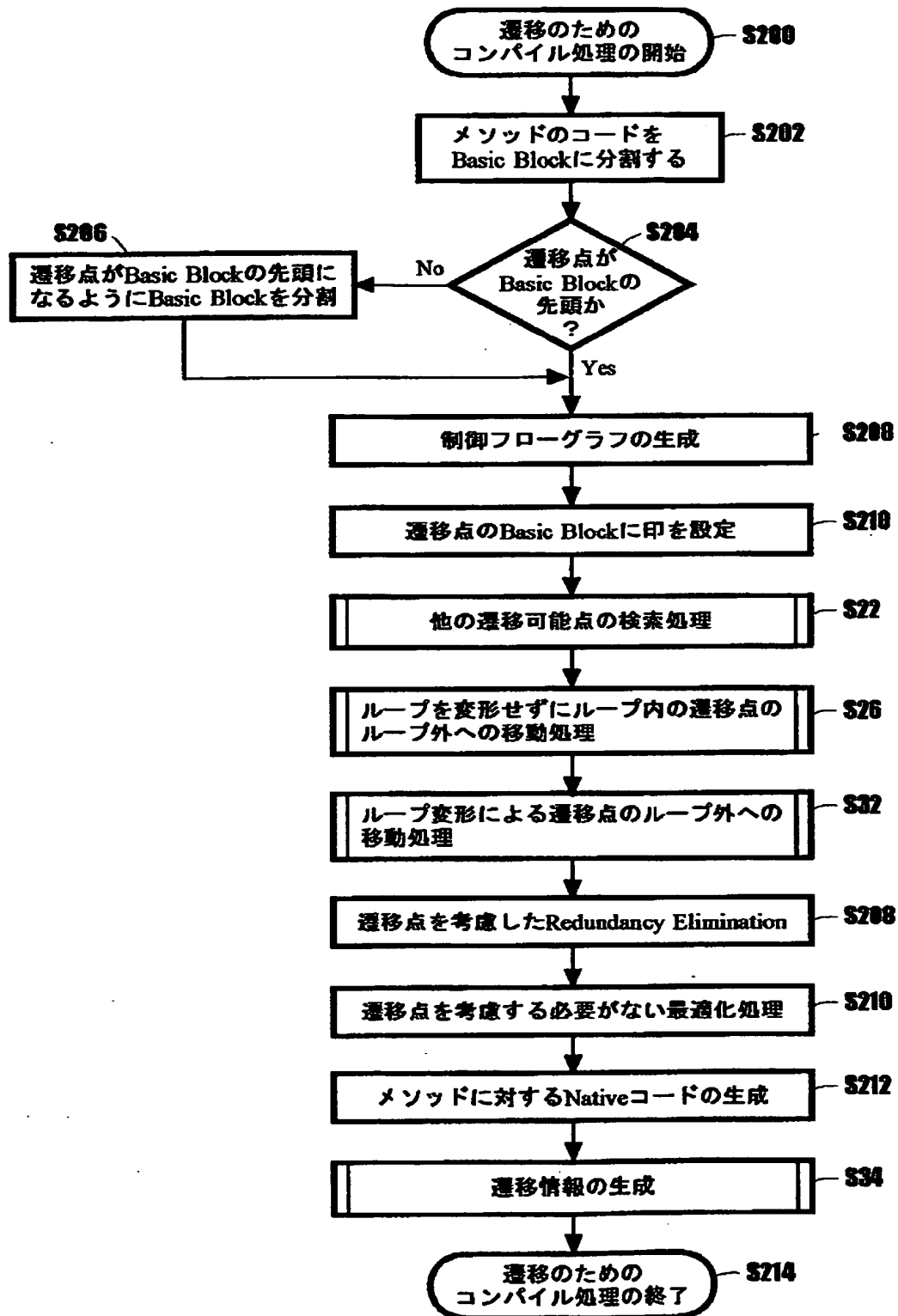


S10

【図 3】

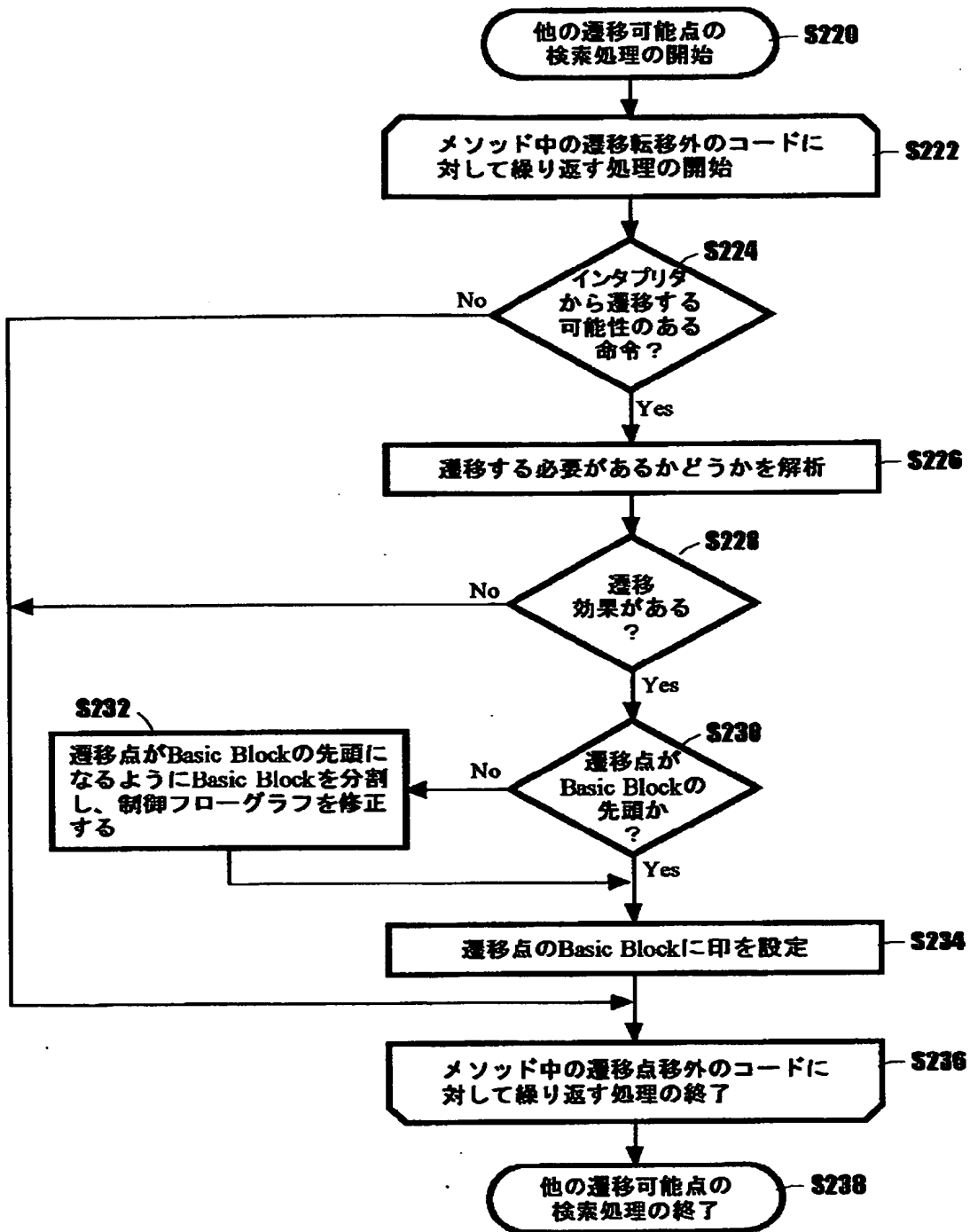


【図 4】



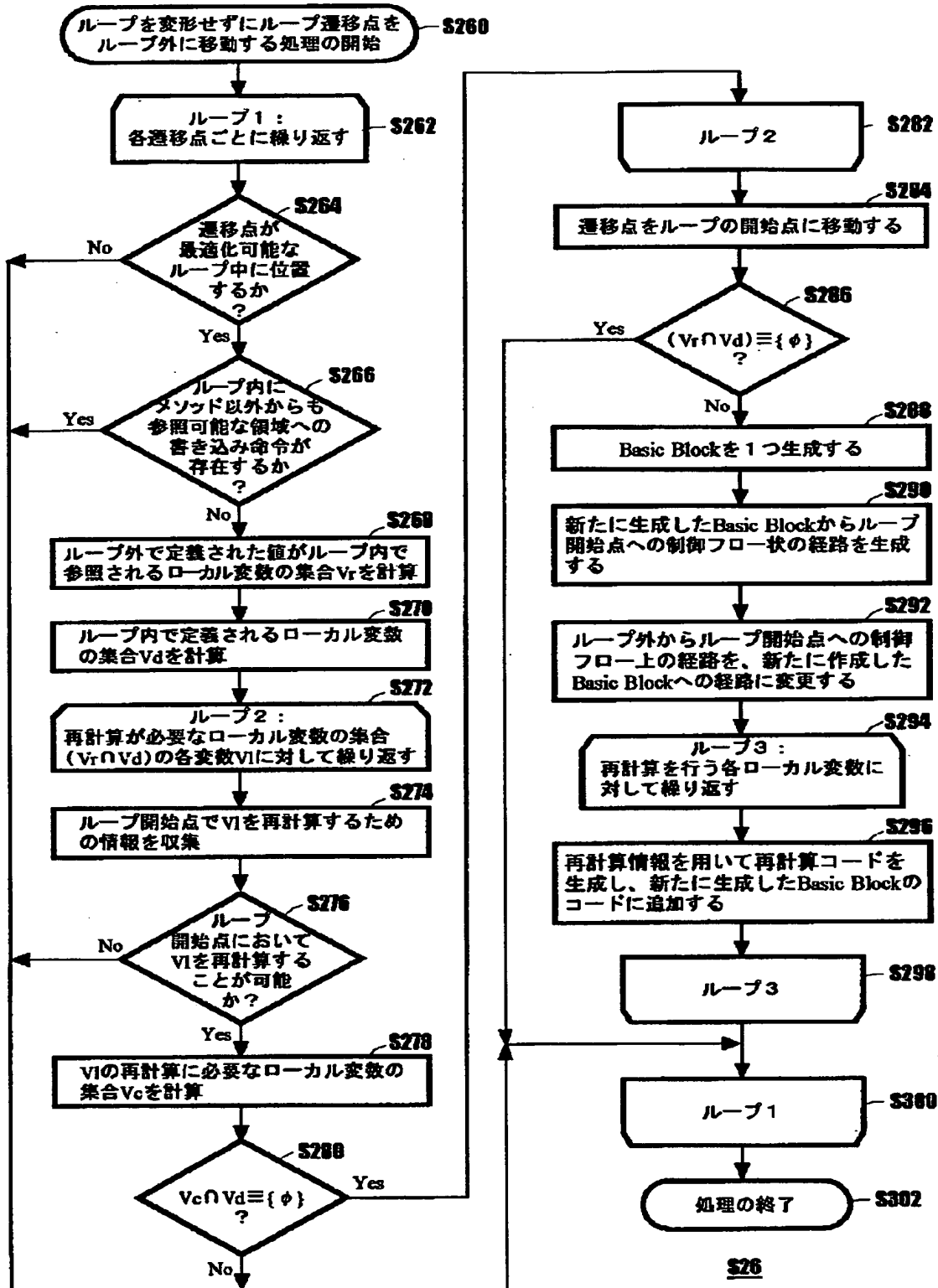
S20

【図 5】

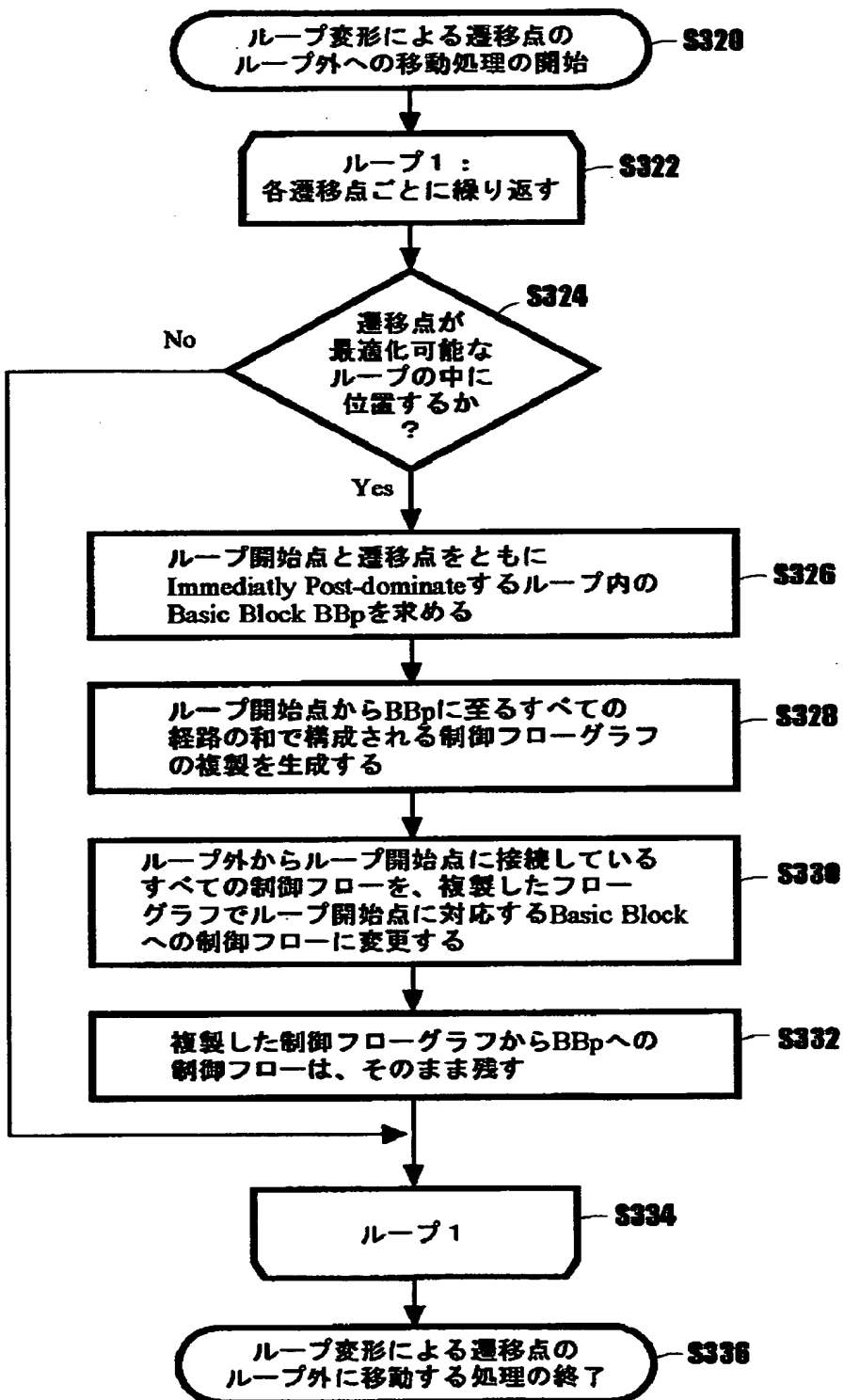


S22

【図 6】

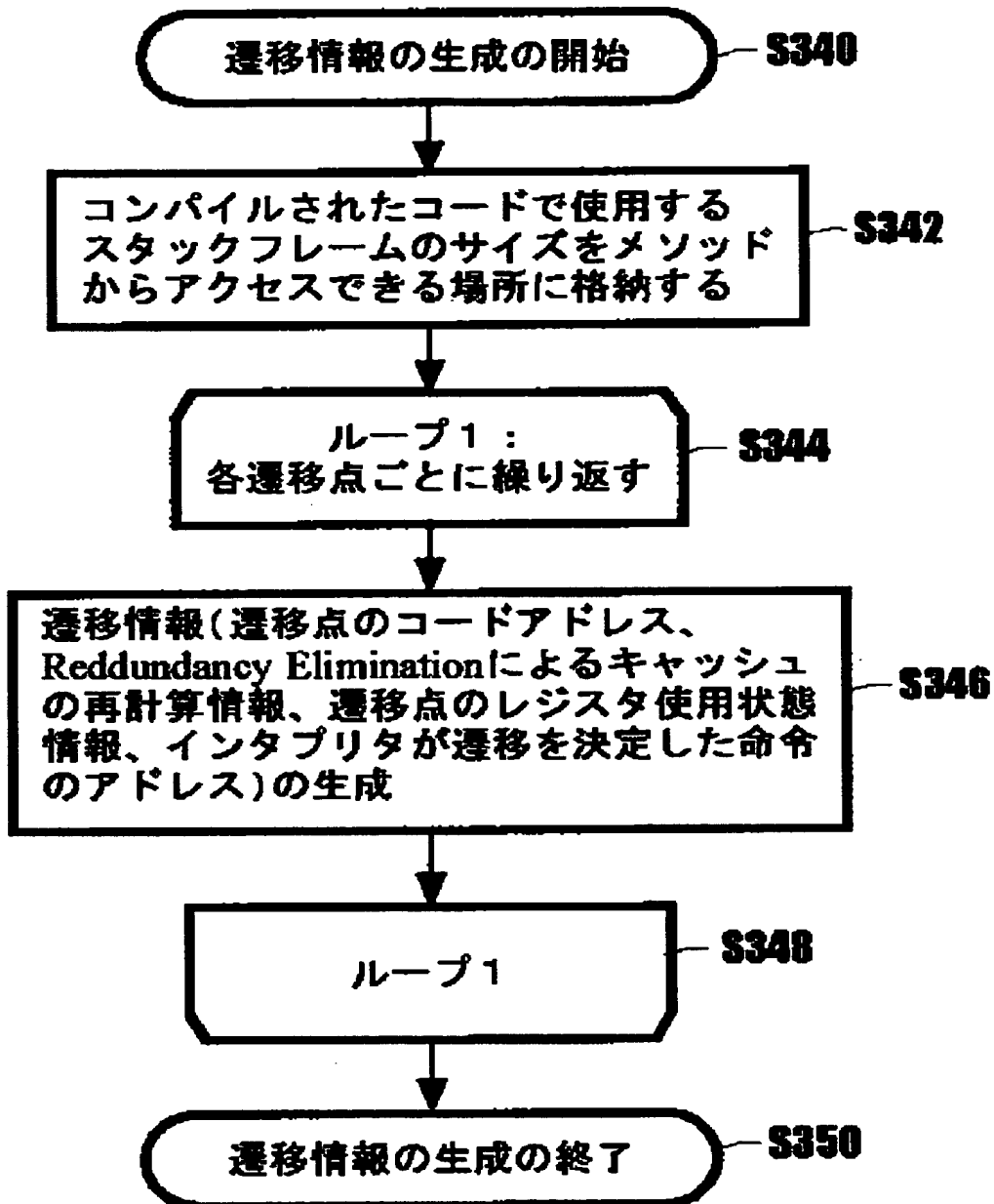


【図 7】



S32

【図 8】



S34

【書類名】 要約書

【要約】

【課題】 より高度な最適化が可能なプログラム実行方法を提供する。

【解決手段】 本発明に係るプログラム実行装置 1 は、メソッドを実行する途中で、コンパイル処理とインタプリタ処理との間で遷移を行う。この際、ループ入り口に遷移点を移動させても実行上の問題がない場合には、コードの遷移点をループの入り口に移動させ、遷移点がループの内部に位置する場合には、ループの入り口と遷移点とをポストドミネートする点をループの直前に複製し、再計算コードを生成するための情報を遷移点に持たせ、再計算を実行する。

【選択図】 図 1

認定・付加情報

特許出願の番号	平成11年 特許願 第326990号
受付番号	59901124793
書類名	特許願
担当官	第七担当上席 0096
作成日	平成11年11月19日

<認定情報・付加情報>

【提出日】	平成11年11月17日
-------	-------------

出 願 人 履 歴 情 報

識別番号 [390009531]

1. 変更年月日 1990年10月24日

[変更理由] 新規登録

住 所 アメリカ合衆国10504、ニューヨーク州 アーモンク (番地なし)

氏 名 インターナショナル・ビジネス・マシーンズ・コーポレイション